Finding Heavy Paths in Graphs: A Rank Join Approach

Mohammad Khabbaz Smriti Bhagat Laks V.S. Lakshmanan

University of British Columbia
Vancouver, BC, Canada
{mkhabbaz, smritib, laks}@cs.ubc.ca

ABSTRACT

Graphs have been commonly used to model many applications. A natural problem which abstracts applications such as itinerary planning, playlist recommendation, and flow analysis in information networks is that of finding the heaviest path(s) in a graph. More precisely, we can model these applications as a graph with non-negative edge weights, along with a monotone function such as sum, which aggregates edge weights into a path weight, capturing some notion of quality. We are then interested in finding the top-k heaviest simple paths, i.e., the k simple (cycle-free) paths with the greatest weight, whose length equals a given parameter ℓ. We call this the *Heavy Path Problem* (HPP). It is easy to show that the problem is NP-Hard.

In this work, we develop a practical approach to solve the Heavy Path problem by leveraging a strong connection with the well-known Rank Join paradigm. We first present an algorithm by adapting the Rank Join algorithm. We identify its limitations and develop a new exact algorithm called HeavyPath and a scalable heuristic algorithm. We conduct a comprehensive set of experiments on three real data sets and show that HeavyPath outperforms the baseline algorithms significantly, with respect to both ℓ and k. Further, our heuristic algorithm scales to longer lengths, finding paths that are empirically within 50% of the optimum solution or better under various settings, and takes only a fraction of the running time compared to the exact algorithm.

1. INTRODUCTION

With increasing availability of data on real social and information networks, the past decade has seen a surge in the interest in mining and analyzing graphs to enable a variety of applications. For instance, the problem of finding dense subgraphs is studied in the context of finding communities in a social network [7], and the Minimum Spanning Tree (MST) problem is used to find teams of experts in a network [12]. Several real applications can be naturally modeled using a problem that we call the *Heavy Path Problem* (HPP): given a weighted graph as input, find top-k heaviest simple (i.e., cycle-free) paths of length ℓ, where the weight of a path is defined as a monotone aggregation (e.g., sum) of weights of the edges that compose the path. Surprisingly, this problem has received relatively less attention in the database community.

We present a few concrete applications of HPP below.

(1) In [8], Hansen and Golbeck motivate a class of novel applications that recommend collections of objects, for instance, an application that recommends music (or video) playlists of a given length. They emphasize three key properties for desirable lists: value of individual items (say songs), co-occurrence interaction effects, and order effects including placement and arrangement of items. We can abstract a graph from user listening history, where a node represents a song of high quality as determined by user ratings and an edge between a pair of songs exists if they were listened to together in one session. The weight on an edge represents how frequently the songs were listened together by users. Heavy paths in such a graph correspond to playlists of high quality songs that are frequently enjoyed together.

(2) Consider an application that recommends an itinerary for visiting a given number of popular tourist spots or points of interest (POIs) in a city [3]. The POIs can be modeled as nodes in a graph, and an edge represents that a pair of POIs is directly connected by road/subway link. The weight on an edge is a decreasing function of the travel time (or money it costs) to go from one POI to another. Heavy paths in such a graph correspond to itineraries with small overall travel time (or cost) for visiting a given number of popular POIs.

(3) Say we want to analyze how a research field has evolved. Given a citation network of research papers and additional information on the topics associated with papers (possibly extracted from the session title of the conference, or from keywords etc.), we can abstract a "topic graph". Nodes in a topic graph represent topics, and an edge between two topics exists if a paper belonging to one topic cites a paper belonging to another topic. The weight on an edge may be the normalized frequency of such citations. Heavy paths in such a graph capture strong flows of ideas across topics.

In addition, Bansal et al. [2] modeled the problem of identifying temporal keyword clusters in blog posts (called Stable Clusters Problem in [2]) as that of finding heavy paths in

a graph. In all these applications, we may be interested in not just the top-1 path, but top-k heavy paths, for instance, k itineraries or playlists to choose from. Finding the top-k heavy paths of a given length ℓ is not trivial. The setting studied in [2] is a special case of the HPP where the input graph was ℓ-partite (each partition corresponds to a timestamp) and acyclic. They proposed three algorithms based on BFS (breadth first search), DFS (depth first search) and the TA (threshold algorithm). Due to the special structure of their graph, i.e., the absence of cycles and only a subset of nodes acting as starting points for graph traversal, they were able to efficiently adapt the BFS and DFS algorithms and show that these adaptations outperform a TA-based algorithm. Our problem is more general in that the graphs are *not* ℓ-partite, and typically contain many cycles. Adaptations of the algorithms proposed in [2] lead to expensive solutions. For example, an adaptation of DFS would require performing ℓ-deep recursion starting at each node, which is prohibitively expensive for large general graphs.

We present an efficient algorithm for HPP called HeavyPath, based on the Rank Join paradigm, as detailed in Section 3.2. In the last decade, there has been substantial amount of work on Rank Join [10, 11, 13, 16, 17]. However, the experimental results reported have confined themselves to Rank Join over a small number of relations. In the driving applications mentioned above, there is often the need to discover relatively long heavy paths. For example, in playlist recommendation, a user may be interested in getting a list containing several tens of songs, and in itinerary planning, recommendations consisting of 5-15 POIs for the tourist to visit within a given time interval (e.g., a day or a week) are useful. In general, *it is computationally more demanding to find top-k heaviest paths as the path length increases. There is a need for efficient algorithms for meeting this challenge.*

The approach we develop in this paper is able to scale to longer lengths compared with classical Rank Join. By exploiting the fact that the relations being joined are identical, we are able to provide smarter strategies for estimating the requred thresholds, and hence terminate the algorithm sooner than classical Rank Join. In addition, we carefully make use of random accesses in the context of Rank Join as an additional means of ensuring that heavy paths are discovered sooner, and the thresholds are aggressively lowered. Finally, it turns out that all exact algorithms considered (including Rank Join and our proposed HeavyPath algorithm) run out of memory when the length of the desred path exceeds 10 on some data sets. As we will see in Section 2, HPP is NP-hard. In order to deal with this, we develop a heuristic approach that works with the allocated memory and allows us to estimate the distance to the optimum solution for a given problem instance. We empirically show that this heuristic extension scales well even for paths of length 100 on real data sets.

We make the following contributions in the paper:

- We formalize the problem of finding top-k heavy paths of a given length for general graphs, and establish the connection between HPP and the Rank Join framework for constructing heavy paths (Section 2).
- We present a variety of exact algorithms, including two baselines obtained by adapting known algorithms, a simple adaptation of Rank Join for computing heavy paths, and an efficient adaptation of Rank Join called HeavyPath. With simple modifications, we can turn HeavyPath into a heuristic algorithm with the nice property that we can derive an empirical approximation ratio in a principled manner (Sections 3, 4, and 5).
- We present a comprehensive set of experiments to evaluate and compare the efficiency and effectiveness of the different algorithms on three real datasets: last.fm, Cora, and Bay that respectively model the three motivating applications described earlier. Our results show that HeavyPath is orders of magnitude faster than the baselines and Rank Join, and can find exact solutions for paths that are several hops longer as compared with all other algorithms. In addition, our heuristic algorithm finds paths that are empirically within 50% of the optimum solution or better under various settings, while taking a fraction of the running time compared to the exact algorithm (Section 6).

We review the related work in Section 7 and conclude in Section 8.

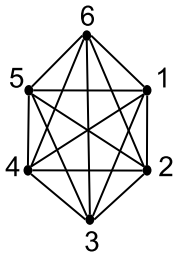
2. PROBLEM STUDIED

Given a weighted graph $G(V, E, W)$, where weight $w_{(u,v)} := W(u, v)$ represents the non-negative weight[1] on edge $(u, v) \in E$, and parameters k and ℓ, the *Heavy Path Problem (HPP)* is to find the top-k heaviest simple paths of length ℓ, i.e., k simple paths of length ℓ with the highest weight. A simple path of length ℓ is a sequence of nodes $P = (v_0, \ldots, v_\ell)$ such that $(v_i, v_{i+1}) \in E$, $0 \leq i < \ell$, and there are no cycles, i.e., the nodes v_i are distinct. Unless otherwise specified, in the rest of the paper, we use the term path to mean simple path. We note that our framework allows path weights defined using any monotone aggregate function of edge weights. For simplicity, we define the weight of a path $P = (v_0, ..., v_\ell)$, as $P.weight = \sum_{j=0}^{\ell-1} w_{(j,j+1)}$.

We note that the heavy path problem for a given parameter ℓ is equivalent to the well-known ℓ-TSP[2] (ℓ-Traveling Salesperson) problem [1], defined as follows: Given a graph with non-negative edge weights, find a path of minimum weight that passes through any $\ell + 1$ nodes. It is easy to see that for a given length ℓ, a path P is a solution to ℓ-TSP iff it is a solution to HPP (with $k = 1$) on the same graph but with edge weights modified as follows: let $w_{(u,v)}$ be the weight of an edge (u, v) in the ℓ-TSP instance, and w_{max} be the maximum weight of any edge in that instance; then the edge weight in the HPP instance is $w'_{(u,v)} = 1 - w_{(u,v)}/w_{max}$. It is well-known that TSP and ℓ-TSP are NP-hard and the reduction above shows HPP is NP-hard as well. In general, HPP can be defined for both directed and undirected

[1]Non-negativity is not a requirement, but is intuitive in most applications.

[2]In the literature it is called k-TSP, where k is the given length of the path. We refer to it as ℓ-TSP to avoid confusion with the parameter k used for number of paths in our top-k setting.



(a)

Edge	Edge Weight
(1,2)	0.93
(2,3)	0.93
(1,3)	0.87
(2,4)	0.77
(1,6)	0.76
(2,5)	0.73
(3,4)	0.73
(1,4)	0.73
(5,6)	0.72
(3,5)	0.70
(1,5)	0.70
(1,6)	0.70
(4,5)	0.69
(3,6)	0.66
(4,6)	0.58

(b)

Figure 1: Example graph for playlist recommendation, and corresponding edge weights table

graphs. Our algorithms and results focus on the undirected case, but can be easily extended to directed graphs.

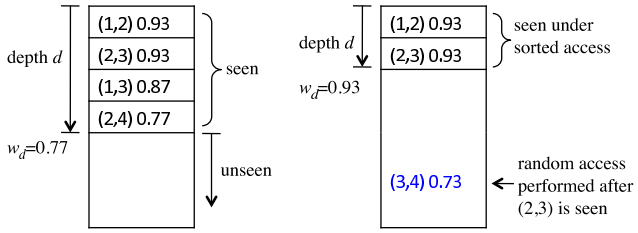
We next briefy discuss the utility of heavy paths for applications, compared with cliques. Some data sets like one that represents the road network (see Bay dataset in Section 6) naturally do not exhibit large cliques. On such data sets, heavy paths order the items (i.e., nodes) in a sequence determined by the graph structure. On the other hand, some data sets may contain large cliques. Figure 1 shows an example of the co-listening graph for the playlist recommendation application (this is a subgraph induced over 6 songs from the last.fm dataset used in our experiments – see Section 6). Say the task is to find heavy paths of length $\ell = 4$. Even when the graph is a clique and any permutation of 5 nodes will result in a path of length 4, the order in which the nodes are visited can make a significant difference to the overall weight of the path. Here, the heaviest simple path of $\ell = 4$ is obtained by visiting nodes in the order 6–1–2–3–4 has a weight of 3.35. In contrast, a different permutation of the nodes 4–3–6–1–2 has a weight of 3.08.

3. FINDING HEAVY PATHS

3.1 Baselines

An obvious algorithm for finding the heaviest paths of length ℓ is performing a depth-first search (DFS) from each node, with the search limited to a depth of ℓ, while maintaining the top-k heaviest paths. This is an exhaustive algorithm and is not expected to scale. A somewhat better approach is dynamic programming. Held and Karp [9] proposed a dynamic programming algorithm for TSP, that works with a notion of "allowed nodes" on paths between pairs of nodes, which we adapt to HPP as follows. For a set of nodes S, we say there is an S-avoiding path from node x to y provided none of the nodes on this path other than x, y are in the set S. E.g., the path $(1, 2, 3)$ is $\{1, 4, 5\}$-avoiding but not $\{2, 4\}$-avoiding. The idea is to find the heaviest simple path of length ℓ ending at j for every node j, and then find the heaviest among them. To find the heaviest simple path ending at j, we find the heaviest simple $\{j\}$-avoiding path of length ℓ that ends at j.

The heaviest path of length 1 (starting anywhere) and ending at a given node j is simply the heaviest edge ending at j. In general, the heaviest (simple) S-avoiding path of length $2 \leq l \leq \ell$ ending at j is found by picking the heaviest among the following set of combinations: concatenate the heaviest $S \cup \{y\}$-avoiding path (from anywhere) of length $l-1$ to any neighbor y of j where $j \in S$ and $y \notin S$, with the edge (y, j).



(a) No random access (b) With random access

Figure 2: Adapting Rank Join for HPP

We can apply this idea recursively and easily extend the dynamic program for finding top-k heaviest paths of a given length. We skip the details for brevity. The equations and details of the dynamic programming algorithm can be found in Appendix A. Clearly, both dynamic programming and DFS have an exponential time complexity, although unlike DFS, dynamic programming aggregates path segments early, thus achieving some pruning. Both algorithms are used as baselines in this paper.

3.2 Rank Join Algorithm for HPP

The methods discussed above are unable to prune paths that have no hope of making it to the top-k. Rank Join [10, 17] is an efficient algorithmic framework specifically designed for finding top-k results of a join query. We briefly review it and discuss how it may be adapted to the HPP problem.

Background. We are given m relations $R_1, ..., R_m$. Each relation contains a *score* attribute and is sorted on non-ascending score order. The score of each tuple in the join $R_1 \bowtie \cdots \bowtie R_m$ is defined as $f(R_1.score, \ldots, R_m.score)$ where $R_i.score$ denotes the score of the current tuple in R_i and f is a monotone aggregation function. The problem is to find the top-k join results for a given parameter k. The Rank Join algorithm proposed by Ilyas et al. [10] works as follows. (1) Read tuples in sorted order from each relation in turn. This is called *sorted access*. (2) For each tuple read from R_i, join it with all tuples from other relations read so far and compute the score of each result. Retain the k result tuples with the highest score. (3) Let d_i be the number of tuples read from R_i so far and let t_i^j be the tuple at position j in R_i. Define a threshold $\theta := \max\{f(t_1^{d_1}.score, t_2^1.score, \ldots, t_m^1.score), \ldots, f(t_1^1.score, \ldots, t_i^{d_i}.score, \ldots, t_m^1.score), \ldots, f(t_1^1.score, \ldots, t_i^1.score, \ldots, t_m^{d_m}.score)\}$. This threshold is the maximum possible score of any future join result. (4) Stop when there are k join results with score at least θ. It is clear no future results can have a score higher than that of these k results.

In the rest of the paper, we will assume f is the sum function. Our results and algorithms carry over to any monotone aggregation function.

Adapting Rank Join for HPP. Given a weight sorted table E of edges, HPP can be solved using the Rank Join framework. Indeed, paths of length ℓ can be found via an ℓ-way self-join of the table E, where the join condition can ensure the end node of an edge matches the start node of the

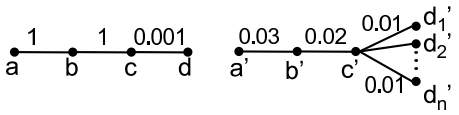


Figure 3: Example instance of HPP. Graph has one heavy path and n lighter paths.

next edge and cycles are disallowed. In particular, whenever a new edge is seen from E (under *sorted access*), it is joined with every possible combination of $\ell - 1$ edges that have already been seen to produce paths of length ℓ. Only the k paths with the highest weight are retained in the buffer. Let d be the depth of the edge (tuple) last seen under sorted access and let w_d denote the weight of the edge seen at depth d and w_{max} be the maximum weight of any edge. The threshold is updated as $\theta := w_d + (\ell - 1)w_{max}$. We stop when k paths of length ℓ are found with weight at least θ. For simplicity, we refer to this adapted Rank Join algorithm as just Rank Join.

Consider the example graph in Figure 1. A Rank Join performed on 4 copies of the edge weight table, with an appropriately defined join condition can be used to compute top-k heavy paths of length $\ell = 4$. The Rank Join algorithm proceeds by scanning the edge table in sorted order of the edge weights. Figure 2(a) illustrates a snapshot during the execution, where the algorithm has "seen" or scanned 4 tuples from the edge table, and therefore the depth $d = 4$. Also, the weight of edge seen at depth 4 $w_d = 0.77$ and $w_{max} = 0.93$. At this depth, the threshold is updated as $\theta = 0.77 + (4 - 1) * 0.93 = 3.56$. For this particular example, Rank Join scans all edges in the table before being able to output the top-1 path of $\ell = 4$ as the weight of the top path (i.e., 3.35) is not over the threshold $\theta = 0.58 + (4 - 1) * 0.93 = 3.37$ even at depth $d = |E|$.

4. LIMITATIONS AND OPTIMIZATIONS

In this section, we make some observations about the limitations of using Rank Join for finding top-k heavy paths in general graphs and discuss possible optimizations. We establish some key properties of (the above adaptation of) Rank Join, which will pave the way for a more efficient algorithm in Section 5.

OBSERVATION 1. *Let P be a path of length ℓ and suppose e is the lightest edge on P, i.e., its weight is the least among all edges in P. Then until e is seen under sorted access, the path P will not be constructed by Rank Join.*

We illustrate this observation with an example graph in Figure 3 for the task of finding the heaviest path of length 3. The graph consists of one path of weight 2.001 (the top path) and n other paths of weight $0.03 + 0.02 + 0.01 = 0.06$, all of length 3. Clearly, the top path (call it P) is the unique heaviest path of length 3. However, Rank Join must wait for the edge with weight 0.001 (the lightest in the graph) to be seen before it can find and report P. Until then, it would be forced to produce (and discard!) the n paths of length 3, each with weight 0.06. Since n can be arbitrarily large, Rank Join is forced to construct an arbitrarily large number of paths most of which are useless. Indeed, as an extreme case, we can show the following result on Rank Join, for the special case $k = 1$.

LEMMA 1. *Consider an instance of HPP where the weight of the heaviest path of length ℓ is smaller than $w_{min} + (\ell - 1) \times w_{max}$, where w_{max} and w_{min} are the weights of the heaviest and lightest edge in the graph, respectively. For this instance, Rank Join produces every path of length ℓ.*

PROOF. The Rank Join algorithm stops when there is a path of length ℓ in the buffer whose weight is not smaller than the threshold. Assume the heaviest path P is lighter than $w_{min} + (\ell - 1) \times w_{max}$, and Rank Join stops after seeing an edge e with weight $w \geq w_{min}$. In this case, the threshold is $\theta = w + (\ell - 1)w_{max} \geq w_{min} + (\ell - 1)w_{max} > P.weight$, so by definition, Rank Join cannot terminate, a contradiction! This shows, it must see all edges with weight w_{min} before halting. By this time, by definition, Rank Join will have produced all paths of length ℓ. □

The observation and lemma motivate the following optimization for finding heaviest paths.

OPTIMIZATION 1. *We should try to avoid delaying the production of a path of a certain length until the lightest edge on the path is seen under sorted access. One possible way of making this happen is via random access. However, random accesses have to be done with care in order to keep the overhead low.*

Figure 2(b) shows that an edge (3,4) that joins with the edge (2,3) can be accessed sooner by using random access. It is worth noting that Ilyas et al. [10] mention the value of random access for achieving potentially tighter thresholds. Of course, the cost of random access is traded for the pruning power of the improved threshold. Indeed, in recent work Martinenghi and Tagliasacchi [14] study the role of random access in Rank Join from a cost-based perspective.

Following this optimization, suppose we use random accesses to find "matching" edges with which to extend heaviest paths of length $\ell - 1$ to length ℓ. This is a good heuristic, but heaviest paths of length $\ell - 1$ may not always lead to heaviest paths of length ℓ. A natural question is whether the use of random accesses in this manner will lead to a performance penalty w.r.t. Rank Join. Our next result shows that Rank Join will produce heaviest paths of length $\ell - 1$ before it produces and reports the heaviest path of length ℓ.

LEMMA 2. *Run Rank Join on an instance of HPP first with input length $\ell - 1$, and then with input length ℓ. Suppose d' (resp., d) is the depth at which Rank Join finds the heaviest path of length $\ell - 1$ (resp., ℓ) for the first time in the respective runs, then, $d \geq d'$. Thus, by the time Rank Join produces the heaviest path of length ℓ, it also produces the heaviest path of length $\ell - 1$.*

PROOF. Suppose P is the heaviest path of length ℓ and Q is the heaviest path of length $\ell - 1$[3]. Suppose $w_{d'}$ and w_d are

[3] Ties are broken arbitrarily.

the edge weights at depths d' and d resp. in the edge table E. Assume $d < d'$. We know that $w_{d'} + (\ell - 2) \times w_{max} \leq Q.weight \leq w_{d'-1} + (\ell - 2) \times w_{max}$ and $w_d + (\ell - 1) \times w_{max} \leq P.weight$. Therefore, $P.weight - w_{max} \geq w_d + (\ell - 2) \times w_{max} \geq w_{d'-1} + (\ell - 2) \times w_{max} \geq Q.weight$. Now, $P.weight - w_{max}$ is a lower bound on the heaviest sub-path of P of length $\ell - 1$. This means P has a sub-path of length $\ell - 1$ that is at least as heavy as Q, and can be returned by Rank Join at depth $d < d'$. This contradicts the fact that Q is the first heaviest path of length $\ell - 1$ discovered by Rank Join. Thus, the heaviest path of length $\ell - 1$ is found no later than the heaviest path of length ℓ. □

Figures 3 and 4 illustrate the notions of depth and edge weight at a given depth. The leftmost table in Figure 4 represents the sorted edge list for the graph in Figure 3. While computing the heaviest path of length $\ell = 3$, Rank Join obtains the heaviest path of $\ell - 1$ (i.e., 2) at depth d'. Without random accesses, the heaviest path of length $\ell = 3$ is obtained after scanning edge (c,d) at depth d.

The real purport of the above lemma is that by the time Rank Join reports a heaviest path of length ℓ, it has already seen heaviest paths of length $\ell - 1$. Thus, any heaviest paths of length $\ell - 1$ we try to extend to length ℓ (using random access) are necessarily produced by Rank Join as well, albeit later than if we were to use no random access.

OBSERVATION 2. *Rank Join uses a very conservative threshold $w_d + (\ell - 1)w_{max}$, by supposing the new edge seen may join with $\ell - 1$ edges with maximum weight. In many cases, the new edge just read may not join with such heavy edges at all.*

One way of making the threshold tighter is by keeping track of shorter paths. For example, if we know P is a heaviest path of length $\ell - 1$, we can infer that the heaviest path of length ℓ cannot be heavier than $P.weight + w_{max}$, a bound often much tighter than $w_d + (\ell - 1)w_{max}$[4]. For this, the (heaviest) paths of length $\ell - 1$ have to be maintained. Pursuing this idea recursively leads to a framework where we maintain heaviest paths of each length i, $2 \leq i \leq \ell$. More precisely, we can perform the following optimization.

OPTIMIZATION 2. *Maintain a buffer B_i for the heaviest paths of length i, and a threshold θ_i on the weight of all paths of length i that may be produced in the future, based on what has been seen so far. When a new heaviest path P of length $i-1$ is found, i.e., $P.weight \geq \theta_{i-1}$, update the threshold θ_i for buffer B_i as $\theta_i = \max(\theta_{i-1}, B_{i-1}.topScore) + w_{max}$ where $B_{i-1}.topScore$ is the weight of the heaviest path in B_{i-1}* after *the current heaviest path P with $P.weight \geq \theta_{i-1}$ is removed from B_{i-1}*[5]*. The rationale is that, in the best case, a heaviest future path of length i may be obtained by joining the current heaviest path in B_{i-1} with an edge that has the maximum weight, or joining a future path in B_{i-1} with such an edge.*

[4]Our algorithm makes use of an even tighter threshold as described shortly.

[5]$B_{i-1}.topScore$ may be greater or less than θ_{i-1}.

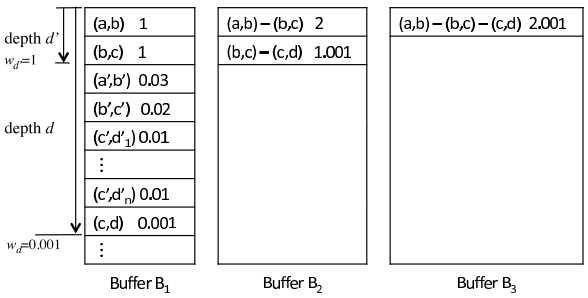


Figure 4: HEAVYPATH **using buffers to extend paths for Figure 3**

Figure 4 schematically describes the idea of using buffers for different path lengths. Buffer B_1 is the sorted edge list for the graph in Figure 3. Buffers B_2 and B_3 store paths of length 2 and 3 respectively. When random accesses are performed to extend paths of length $l - 1$ to those of length l, the buffers can be used to store these intermediate paths. For instance, when the heaviest path of length 1 is seen, say edge (a, b) is seen, it can be extended to paths of length 2 by accessing edges connected to its end points, as represented in B_2 in Figure 4. Similarly, the heaviest path of length 2 can be extended by an edge using random access to obtain path(s) in buffer B_3.

OBSERVATION 3. *Rank Join, especially when performed as a single ℓ-way operation, tends to produce the same sub-path multiple times. For example, when Rank Join is required to produce the heaviest path of length 3 for the graph in Figure 3, for paths (a', b', c', d'_i), $i \in [1, n]$, it produces the length 2 sub-path (a', b', c') n times as it does not maintain shorter path segments.*

This observation motivates the following optimization.

OPTIMIZATION 3. *Shorter path segments can be maintained so they are created once and used multiple times when extended with edges to create longer paths. For instance, in Figure 3, we can construct (and maintain) the length 2 path segment (a', b', c') once and use it n times if needed as edges are appended to extend it into n different length 3 paths.*

A concern surrounding a random access based approach to extending heaviest paths of length $l - 1$ into paths of length l is that it may result in extending too many such paths. Our next result relates these paths to those produced by Rank Join as part of its processing.

LEMMA 3. *Let P be the heaviest path of length ℓ. When Rank Join terminates, every path of length $\ell - 1$ that has weight no less than $P.weight - w_{max}$, will be created.*

PROOF. Suppose Rank Join finds P at depth d. This means $P.weight \geq w_d + (\ell - 1) \times w_{max}$, and, $P.weight - w_{max} \geq w_d + (\ell - 2) \times w_{max}$. Notice that $P.weight - w_{max}$ is a lower bound on the weight of the heaviest path of length $\ell - 1$. Therefore, if there is a path of length $\ell - 1$ that is heavier than $P.weight - w_{max}$, it will be produced by Rank Join by depth d. □

Algorithm 1 HeavyPath (E, ℓ, k)

Input: Sorted edge list E, path length ℓ, number of paths k
Output: top-k heaviest paths of length ℓ

```
for l = 2 to ℓ do
    B_l ← ∅ // empty sorted set
    θ_l = w_max × l
topPaths ← ∅ // empty sorted set
while | topPaths |< k do
    topPaths ← topPaths ∪ NextHeavyPath (E, ℓ)
```

Algorithm 2 NextHeavyPath (E, l)

Input: Sorted list of edges E, and path length l
Output: Next heaviest path of length l

```
1: if l = 1 then
2:    P^1 ← ReadEdge(E)
3:    θ_2 = 2 × P^1.weight
4:    return P^1
5: while B_l.topScore ≤ θ_l do
6:    P^{l-1} ← NextHeavyPath (E, l − 1) // recursion
7:    s, t ← EndNodes(P^{l-1})
8:    for all y ∈ V | (y, s) ∈ E do
9:       B_l ← B_l ∪ ((y, s) + P^{l-1}) // avoiding cycles
10:   for all z ∈ V | (t, z) ∈ E do
11:      B_l ← B_l ∪ (P^{l-1} + (t, z)) // avoiding cycles
12: P^l ← RemoveTopPath(B_l)
13: if l < ℓ then
14:   θ_{l+1} = max(B_l.topScore, θ_l) + w_max
15: return P^l
```

The results, observations, and optimizations discussed in this section suggest an improved algorithm for finding heaviest paths, which we present next.

5. HEAVYPATH ALGORITHM FOR HPP

We start this section by providing an outline of our main algorithm for solving HPP, and subsequently explore some refinements. Our algorithm maintains a buffer B_i for storing paths of length i explored so far, where $2 \leq i \leq \ell$. Let threshold θ_i denote an upper bound on the weight of any path of length i that may be constructed in the future. Each buffer B_i is implemented as a sorted set of paths of length i sorted according to non-increasing weight and without duplication (e.g., paths (a, b, c) and (c, b, a) are considered duplicates). Algorithm 1 describes the overall approach. It takes as input a list of edges E sorted in non-increasing order of edge weights, and parameters ℓ and k. It calls the NextHeavyPath method (Algorithm 2) repeatedly until the top-k heaviest paths of length ℓ are found.

Algorithm 2 describes the NextHeavyPath method. It takes as input a list of edges E sorted in non-increasing order of edge weights, and the desired path length l, $l \geq 2$. It is a recursive algorithm that produces heaviest paths of shorter lengths on demand, and extends them with edges to produce paths of length ℓ. The base case for this recursion is when $l = 1$ and the algorithm reads the next edge from the sorted list of edges. The ReadEdge method returns the heaviest unseen edge in E (sorted access, line 2). If $l < \ell$, the path of length l obtained as a result of the recursion is extended by one hop to produce paths of length $l + 1$. Specifically, a path of length $l < \ell$ is extended using edges (random access, lines 8 and 10) that can be appended to either one of its ends (returned by method EndNodes). The "+" operator for appending an edge to a path is defined in a way that guarantees no cycles are created. The threshold θ_l is updated appropriately and when it becomes smaller than the weight of the heaviest path in buffer B_l, the next heaviest path of length l that is greater than the threshold is returned. This is done by calling the method RemoveTopPath for buffer B_l and returning the resulting path. If $l < \ell$ and the next heaviest path of length l has been obtained, θ_{l+1} is updated.

Updating the thresholds. To start, Algorithm 2 explores the neighborhood of the heaviest edge for finding the heaviest path of length 2. Heavy edges are explored and the threshold θ_2 is updated until the weight of the heaviest explored path of length 2 is greater than θ_2. The NextHeavyPath algorithm updates θ_2 aggressively when the next heaviest edge is seen. It uses the fact that any path of length 2 created later from currently unseen edges cannot have a weight greater than $2 \times P^1.weight$, where P^1 is the lightest edge seen so far. For $l > 1$, θ_{l+1} is updated when the next heaviest path of length l is obtained. Therefore, we can use θ_l as an upper bound on the weight of any path of length l that can be created *in the future* from the buffers B_i, where $i < l$. The maximum of θ_l and the weight of heaviest path in B_l (i.e., $B_l.topScore$) provides an upper bound on the weight of any path of length l that can be created in the future (after the previous heaviest path of length l). Adding w_{max} to the obtained upper bound provides a new (tighter) threshold for paths of length $l + 1$.

THEOREM 1. *Algorithm* HeavyPath *correctly finds top-k heaviest paths of length ℓ.*

Proof. The proof is by induction. The base case is for going from edges to paths of length 2. Given that all of the edges above depth d are extended, the heaviest path that can be created from them is already in B_2. The weight of the heaviest path that can be created from lighter edges is at most $2 \times w_d$. If the heaviest path in B_2 is heavier than $2 \times w_d$, then it must be the heaviest path of length 2.

Assuming the heaviest paths of length l are produced correctly in sorted order, we show the heaviest path of length $l + 1$ is found correctly. Suppose P, the heaviest path of length $l + 1$, is created for the first time from by extending Q, which is the n^{th} heaviest path of length l. The next heaviest path of length l is either already in B_l or has not been created yet. Therefore, $\max(\theta_l, B_l.topScore)$ is an upper bound on the next heaviest path of length l that has not been extended and any edge that can join this path can have weight at most w_{max}. Suppose when the m^{th} heaviest path of length l is seen, $\max(\theta_l, B_l.topScore) + w_{max}$ is updated to a value smaller than $P.weight$. It is guaranteed that P is already in B_{l+1} and has the highest weight in that buffer. In other words, when the threshold is smaller than $P.weight$, the difference between the weight of P and next heaviest path of length l is more than w_{max}. Now, paths of length $l+1$ that can be created from heavier paths of length l are already in the buffer, and no unseen path of length

l can be extended to create a path heavier than P. Therefore, P is guaranteed to be the heaviest path. The preceding arguments hold for top-k heaviest paths where $k > 1$. □

Algorithm 2 extends the heaviest paths in sorted order, to avoid their repeated creation. However, since paths are extended by random accesses, it is possible to create a path twice, which is unnecessary. For instance, a path of length l may be created while extending its heaviest sub-path and again while extending its lightest subpath of length $l-1$.

5.1 Duplicate Minimization by Controlling Random Accesses

Algorithm NEXTHEAVYPATH extends a path of length l to one of length $l+1$ by appending all edges that are incident on either end node of the path. Since these edges are not accessed in any particular order, in the literature of top-k algorithms, they are referred to as random accesses. In this section, we develop a strategy for controlling random accesses performed by Algorithm NEXTHEAVYPATH for minimizing duplicates. Duplicate paths of length $l+1$ can be created either due to extending the same path of length l, or by extending two different subpaths of the same path of length $l+1$. Our solution for avoiding duplicates of the first kind is implementing every buffer as a sorted set. This avoids propagation of duplicates during execution. Further, the threshold update logic guarantees that if there is a copy P' of some path P that is already in the buffer B_l, the path will not be returned before its copy P' makes it to B_l. The algorithm ensures that when P is returned, P' either does not exist or has been constructed and eliminated.

In addition to eliminating duplicates that have been created, we take measures to reduce their very creation.Suppose P is a path of length $l+1$ whose right sub-path of length l is the heaviest path of length l and its left sub-path of length l is the second heaviest path of length l. Since random accesses are performed at both ends of a path, P will be created twice, using each of the top-2 paths of length l.

One possible solution is to perform random accesses at one of the ends of a path. Although this prevents duplicate creation of P, it does not allow fully exploring the neighborhoods of heavier edges. For example, consider a path with edges $\{(a,b),(b,c),(c,d)\}$ that is the heaviest path of length 3, with (b,c) the heaviest in the graph. If the addition of edges is restricted to the beginning of the path, the construction of the heaviest path of length 3 will be delayed until (a,b) is observed. There can be graph instances for which this can happen at an arbitrary depth. Therefore, it is advantageous to extend paths on both ends, and we dismiss the idea of one sided extension.

LEMMA 4. *Suppose Q is the n^{th} heaviest path of length l with (a,b) as its heaviest edge. No new path of length $l+1$ can be created from Q by adding an edge which is heavier than (a,b) using the* NEXTHEAVYPATH *method.*

PROOF. Let P be a new path of length $l+1$ created from Q. If P is derived for the first time, it can not have a subpath of length l that is heavier than Q. Otherwise, the heavier subpath is one of the $n-1$ paths created before Q. On the other hand, adding any edge to the end of Q which is heavier than (a,b) results in a path of length $l+1$ that has a subpath of length l heavier than Q. The lemma follows. □

Therefore, no new path of length $l+1$ can be created by adding an edge to Q, that is heavier than the heaviest edge of Q. This leads to the following theorem.

THEOREM 2. *Using* NEXTHEAVYPATH*, every new path of length $l+1$ is created only by extending its heavier sub-path of length l. No path is created more than twice. A path of length $l+1$ is created twice iff both its sub-paths of length l have the same weight.* □

The strategy for controlling random accesses embodied in Theorem 2 can be generalized to a stronger strategy as follows.

FACT 1. *Using* NEXTHEAVYPATH*, given a path of length l, no new path of length $l+1$ can be created by adding an edge to its rightmost node that is heavier that its leftmost edge, or adding an edge to its leftmost node that is heavier than its rightmost edge.*

In the rest of the paper, we refer to the strategy for controlling random accesses described in Fact 1 as *random access strategy*. Notice that Algorithm HEAVYPATH always employs random access in addition to sorted access. Additionally, we have the option of adopting (or not) the random access strategy above for controlling when and how random accesses are used. We have:

FACT 2. *If all of the edge weights in the graph are distinct, every path is created only once when the random access strategy mentioned above is followed.*

In the rest of this paper, we follow the random access strategy of Fact 1 for performing random accesses unless otherwise specified. We refer to this way of performing random accesses as *random access strategy*. In our experiments, we measure the performance of HEAVYPATH both without and with this strategy.

5.2 HeavyPath Example

In this section, we present a detailed example that illustrates the creation of paths by HEAVYPATH and Rank Join for finding the heaviest path (i.e., $k=1$) of length 3 for the graph in Figure 3. The idea of using buffers was already illustrated in Figure 4, to which we refer below. Later in Section 6, we present several examples of heavy paths found by our algorithms for the applications of finding playlists and topic paths.

HEAVYPATH first reads the first heaviest edge (a,b) and then extends it using a random access to edge (b,c) into the path (a,b,c) of length 2. It then reads edge (b,c) again under sorted access and tries to extend via a random access to

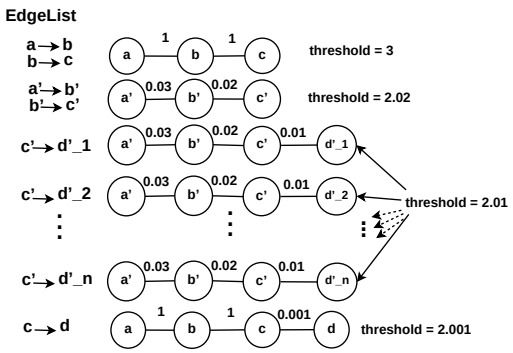


Figure 5: Paths created by Rank Join on Figure 3

(a, b). The duplicate derivation of the path (a, b, c) is caught and discarded. Edge (b, c) is extended with another random access into the path (b, c, d). At this point, paths (a, b, c) of weight 2 and (b, c, d) of weight 1.001 are added to buffer B_2. The threshold θ_2 at this point is 2 and is updated to 0.06 when the next edge (a', b') is visited under sorted access. At this time, the two heaviest paths of length 2 are both above the threshold and are returned. Of these two paths, (a, b, c) is extended with a random access to edge (c, d) to form a length 3 path. If we do not adopt the random access strategy (see Fact 1, Section 5.1), then (b, c, d) will be similarly extended and again the duplicate derivation would be discarded. If we adopt the random access strategy, random access is restricted to edges whose weight is no more than that of the edges at either end of the path, so (b, c, d) will *not* be similarly extended. Now, θ_2 is updated to 2×0.03 and θ_3 is updated to 1.06. The heaviest path of length 3 found so far, which has weight 2.001, is reported.

HeavyPath performs 5 joins in total before reporting the heaviest path of length 3. That includes the joins for paths of length 2 and 3 that are created, and the additional duplicate path that is created and removed during the execution.

Figure 5 illustrates Rank Join for the same graph and parameter settings. Rank Join is not able to produce the heaviest length 3 path until it scans every edge under sorted access for this graph instance. Each edge is joined twice with the partial list of edges that are scanned before it, to construct paths of length 3. Rank Join produces $n+1$ paths of length 3 and two paths of length 2 in the order shown in Figure 5. It performs a total of $2n + 4$ join operations before finding the heaviest length 3 path. This example demonstrates that the performance of Rank Join can be significantly worse than HeavyPath in terms of the number of join operations it performs.

5.3 Memory-bounded Heuristic Algorithm

As described in the earlier sections, the problem of finding the heaviest path(s) is NP-hard. Even though Rank Join uses a fixed buffer space by storing only the top-k heaviest paths at any time, it needs to construct many paths and may run out of allocated memory. On the other hand, HeavyPath explicitly stores intermediate paths in its buffers, and in doing so it may run out of memory (see Section 6 for performance of various algorithms and their memory usage). From a practical viewpoint, having a memory bounded algorithm for HPP would be useful. Thereto, we propose a

Algorithm 3 HeavyPathHeuristic$(\ell, B,)$

Input: Path length ℓ, buffers B from a run of HeavyPath
Output: a heavy path of length ℓ and ratio ρ

```
1: i = LastBufferIndex () // last non-empty buffer
2: j = LastBufferIndex () // last non-empty buffer
3: while i < ℓ do
4:    P^i ← RemoveTopPath(B_i)
5:    s, t ← EndNodes(P^i)
6:    for all y ∈ V | (y, s) ∈ E do
7:       B_{i+1} ← B_i ∪ ((y, s) + P^i) // avoiding cycles
8:    for all z ∈ V | (t, z) ∈ E do
9:       B_{i+1} ← B_i ∪ (P^i + (t, z)) // avoiding cycles
10:   i = LastBufferIndex () // last non-empty buffer
11: q ← ⌊ℓ/(j − 1)⌋
12: r ← ℓ − q × (j − 1)
13: if r > 0 then
14:   U_ℓ ← U_{j−1} × q + U_r
15: else
16:   U_ℓ ← U_{j−1} × q
17: ρ = B_ℓ.topScore/U_ℓ
18: return(RemoveTopPath(B_ℓ), ρ)
```

heuristic algorithm for HPP called HeavyPathHeuristic that takes an input parameter C which is the *total* number of paths the buffers are allowed to hold, in all. If HeavyPath fails to output the exact answer using the maximum allowed collective buffer size C, its normal execution stops and it switches to a greedy post-processing heuristic. Algorithm 3 provides the pseudocode for this heuristic. The post-processing requires only constant additional memory $O(d_{max} \times \ell)$, where d_{max} is the maximum node degree in the graph. This is negligible in most practical scenarios, but should be factored in while allocating memory.

Suppose i is the index of the last non-empty buffer (returned by the routine LastBufferIndex) when HeavyPath runs out of the total allocated buffer size C. At this time, the heaviest path of length $i - 1$ has already been created and extended. However, θ_i is still larger than the weight of the heaviest path in B_i. HeavyPathHeuristic takes the current heaviest path of length i and performs random accesses to create longer simple paths. Note that these random accesses do not follow the rule described in Section 5.1 in order to have a better chance of finding heavier paths. Having done this, it calls LastBufferIndex in a loop. LastBufferIndex returns $i + 1$ if the previous path on the top of B_i has been successfully extended to create at least one path of length $i + 1$. The while loop continues until a path of length ℓ is found. In case none of the existing paths in B_i can be extended to create paths of length ℓ, it *backtracks* to a buffer of shorter path length. The whole process is guaranteed to run in $O(C + d_{max} \times \ell)$ buffer size.

Empirical Approximation Ratio. In addition to returning a path of length ℓ, this heuristic can estimate the worst-case ratio between the weight of the path found and the maximum possible weight of the heaviest path of length ℓ. We call this the *empirical approximation ratio*. Let U_l denote the maximum possible weight of a path of length l. The empirical approximation ratio, denoted ρ, is defined as $\rho = B_\ell.topScore/U_\ell$. If HeavyPath makes it to the j^{th}

Measures	Cora	last.fm	Bay
Nodes	70	40K	321K
Edges	1580	183K	400K
Average Degree	22.6	4.5	1.2
Number of components	1	6534	1

Table 1: Summary of datasets

buffer, the heaviest path of length $l \leq j-1$ has already been found and can be used to provide an approximation ratio along with the output. If j is the last non-empty buffer when HeavyPath terminates, U_l is not known for $l \geq j$. Since the approximation ratio mentioned above is the worst case, we refer to U_l as a pessimistic upper bound on the weight of the heaviest path of length l, for $l \geq j$. The main idea behind this calculation is the following (see lines 11-17): if ℓ is divisible by $j-1$, then no path of length ℓ can be heavier than $U_{j-1} \times (\ell/(j-1))$, since U_{j-1} is an upper bound on the weight of any path of length $j-1$; if ℓ is not divisible by $j-1$, let r be the remainder of the division. By the same reasoning as above, $U_{j-1} \times \lfloor \ell/(j-1) \rfloor + U_r$ is an upper bound on the weight of any path of length ℓ.

6. EXPERIMENTAL ANALYSIS

6.1 Experimental Setup

Algorithms Compared. We implemented the algorithms Dynamic Programing, Rank Join and HeavyPath (without and with the random access strategy). We also implemented the heuristic algorithm HeavyPathHeuristic and a simple Greedy algorithm to serve as a quality baseline for HeavyPathHeuristic. We evaluate our algorithms over three real datasets: Cora, last.fm and Bay, summarized in Table 1. The distributions of edge weights for the three datasets can be found in Figure 6.

Cora. We abstract a topic graph from the Cora Research Paper Classification dataset[6]. Nodes in the topic graph represent research topics into which research papers are classified, and an edge between two topics a, b represents that a paper belonging to topic a cited a paper belonging to topic b or vice versa or both. The weight on an edge is computed as the average of the fraction of citations from papers in topic a to papers in topic b and vice versa. A heavy path in the topic graph captures the flow of ideas across topics.

last.fm. The last.fm data was crawled using their API service[7]. Starting with the seed user "RJ" (Richard Jones was one of the co-founders of last.fm), we performed a breadth first traversal and crawled 400K users. Of these, 163K users had created at least one playlist, for a total of 173K playlists with 1.1M songs in all. We use these playlists as a proxy for listening sessions of users to build a co-listening graph. A node in the co-listening graph is a song and an edge represents that a pair of songs were listened to together in several playlists. The weight of an edge is defined as the Dice coefficient[8]. We filtered out edges that had a dice coefficient smaller than 0.1. The graph obtained has 6534 connected components, which implies that there are many pairs of songs that are not heard together frequently. A heavy path in the co-listening graph captures a popular playlist of songs with a strong "cohesion" between successive songs.

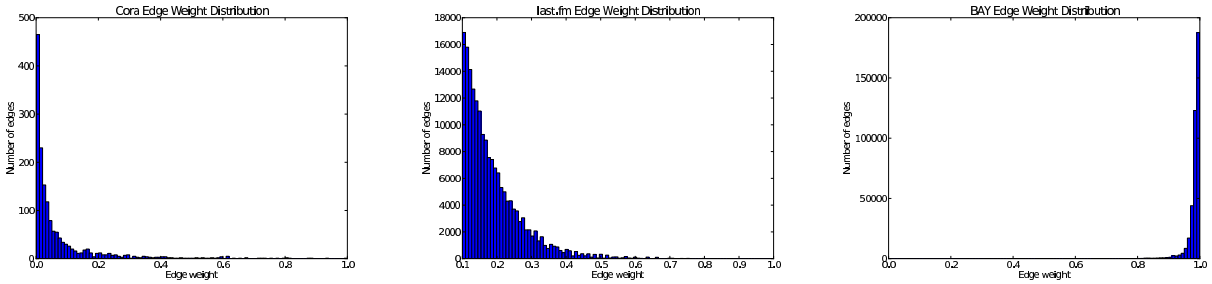
Bay. Our third dataset is a road network graph of the San Francisco Bay area. It is one of the graphs used for the 9th DIMACS Implementation Challenge for finding Shortest Paths[9]. The nodes in the graph represent locations and edges represent existing roads connecting pairs of locations. The weight on an edge represents the physical distance between the pair of locations it connects. We normalize the weights as described in Section 2 to solve the *lightest path problem* on this graph.

Implementation Details. All experiments were performed on a Linux machine with 64GB of main memory and 2.93GHz-8Mb Cache CPU. To be consistent, we allocated 12GB of memory for each run, unless otherwise specified. The algorithms were implemented in Java using built-in data structures (that are similar to priority queues) for implementing buffers as sorted sets. Our implementation of Rank Join was efficient in that it maintained a hash table of scanned edges to avoid scanning the edge list every time a new edge was read under sorted access. Similarly, for efficient random access, we maintained the graph as a sparse matrix.

6.2 Experimental Evaluation

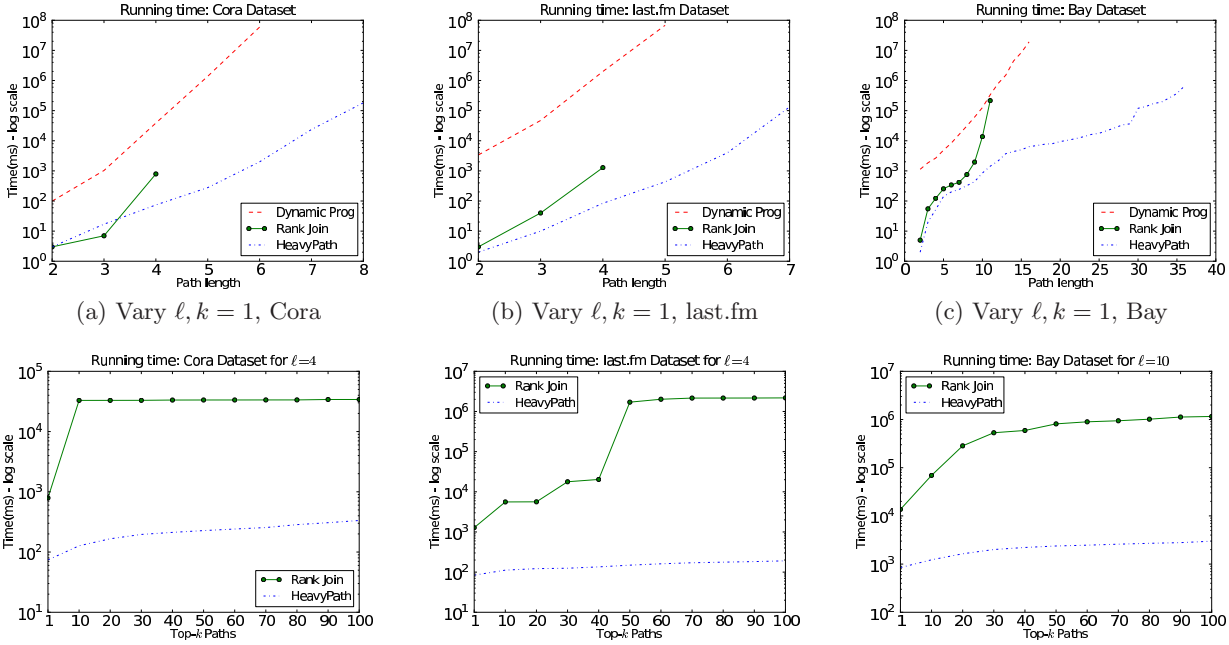
Running time for varying path lengths. We compare the running times of the four algorithms proposed in Section 3 for solving HPP. Figures 7(a), 7(b), 7(c) show the running time for finding the top-1 path of various lengths for Cora, last.fm and Bay datasets respectively. The running time increases with the path length for all algorithms as expected. For all three datasets, the Dynamic Programming algorithm is orders of magnitude slower than the other algorithms even for short paths of length 2. For Cora, Dynamic Programing took over a day to compute the heaviest path of length 6, albeit using only 4GB of the allocated 12GB of memory. In all cases, except for $\ell = 3$ on Cora (see Figure 7(a)), Rank Join is slower than HeavyPath, with the difference in running times of the two methods increasing with path length. We investigated the single instance where Rank Join was faster than HeavyPath and noticed that for short paths that contain high degree nodes, performing random accesses during HeavyPath can result in extra costscompared with Rank Join. After $\ell = 4$ on Cora and last.fm, and $\ell = 11$ on Bay, Rank Join runs out of the allocated memory and quits. In contrast, HeavyPath is able to compute the heaviest paths of length 8 on Cora, 7 on last.fm, and 36 on Bay before running out of the allocated memory. All algorithms are faster and can compute longer paths on Bay as compared with Cora and last.fm. Both the structure of the graph (especially the average degree), and the distribution of the edge weights play a role in the running time and memory usage. The Bay dataset is the largest of the three datasets, but has a small (i.e., 1.2) average node degree, which makes it easier to traverse/construct paths. The Cora dataset has only 70 nodes, but is a dense graph that includes nodes which are connected to all other

[6] `http://www.cs.umass.edu/~mccallum/data`
[7] `http://www.last.fm/api`
[8] $\texttt{dice}(i,j) = \frac{2|i \cap j|}{|i|+|j|}$
[9] `http://www.dis.uniroma1.it/$\sim$challenge9/`



(a) Cora (b) last.fm (c) Bay

Figure 6: Edge weight distributions for the Cora, last.fm and Bay datasets



(a) Vary $\ell, k = 1$, Cora (b) Vary $\ell, k = 1$, last.fm (c) Vary $\ell, k = 1$, Bay

(d) $\ell = 4$, vary k, Cora (e) $\ell = 4$, vary k, last.fm (f) $\ell = 10$, vary k, Bay

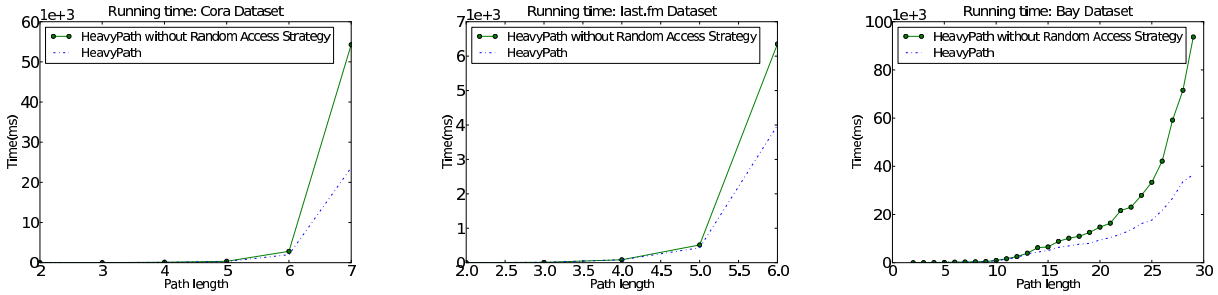
Figure 7: Running time comparisons for exact algorithms with different parameter settings

nodes. In comparison, the last.fm graph has 40K nodes and average node degree 4.5.

On analyzing these results, we make the observation that average node degree and edge weight distribution are the main parameters that define the hardness of an instance. In fact, our smallest dataset in terms of number of nodes and edges, Cora, is the most challenging of all. Polyzotis et al. [17], make similar observations in their experiments on the Rank Join problem.

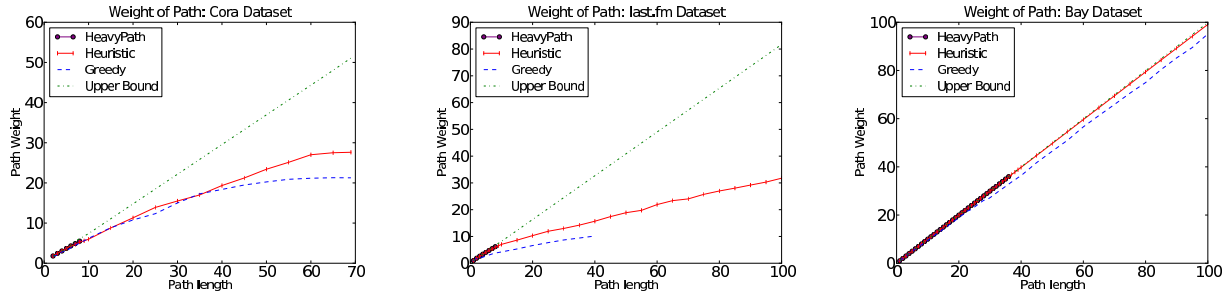
Running time for varying number of top-k paths. Figures 7(d), 7(e), 7(f) show the running times for finding top-k paths of a fixed length, while varying k from 1 to 100. We chose $\ell = 4$ for Cora and last.fm and $\ell = 10$ for Bay as those were the longest path lengths for which all algorithms produced an output within the allocated memory. Since Dynamic Programing is clearly very slow even for $k = 1$, we focus on comparing the other algorithms in this and subsequent experiments. Rank Join shows interesting behavior with increasing k. For smaller values of k, e.g., $k < 50$ for last.fm (see Figure 7(e)), the running time continues to increase as k increases. When k is increased further, the running time does not change significantly. When constructing paths, if the next heaviest path has already been constructed by Rank Join, it can output it immediately, and that seems to be the case for large values of k. Recall that Rank Join may build a subpath many times while constructing paths that subsume it. The increase in running time for HeavyPath as k increases is insignificant. Since the algorithm builds on shorter heavy paths, several paths of length less than ℓ may already be in the buffers and can be extended to compute the next heaviest path.

Impact of Random Access Strategy. Figure 8 compares HeavyPath without and with the random access strategy adopted and shows the additional benefit of employing the random access strategy. Consistent across all datasets, we

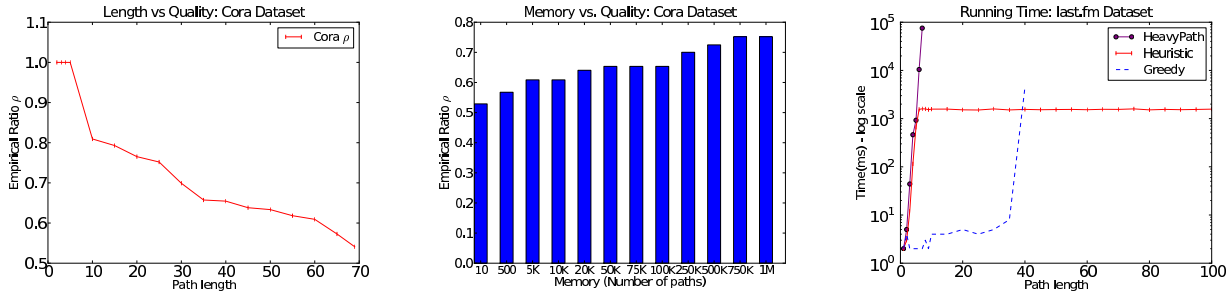


(a) Vary $\ell, k = 1$, Cora (b) Vary $\ell, k = 1$, last.fm (c) Vary $\ell, k = 1$, Bay

Figure 8: Improvement obtained with Random Access Strategy



(a) Path quality using heuristics on Cora (b) Path quality using heuristics on last.fm (c) Path quality using heuristics on Bay



(d) Path length vs. ρ on Cora (e) Memory vs. ρ on Cora (f) Running time of heuristics on last.fm

Figure 9: Performance using Heuristic approach

observe that using random access strategy speeds up the algorithm execution. The speedup is greater for longer paths.

Interpreting heavy paths. We present a few representative example to show the application of HPP to generating playlists and finding flow of ideas, as discussed in Section 1.

The heaviest path returned by HeavyPath on last.fm with $\ell = 5$ has the following songs: "Little Lion Man, Sigh No More, Timshel, I Gave You All, Winter Winds". We noticed that all these songs are from the "folk rock" genre and by the band "Mumford & Sons". Interestingly, the heaviest path of $\ell = 7$ was comprised of a completely different set of songs: "Put It On The Air, Beach House, Substitute For Murder, Souvenir, Ease Your Mind, Novgorod, Sun God, Nemo" from the "art rock" genre, by the band "Gazpacho". On investigating further, we noticed a trend in how users of last.fm create playlists, which is to add a collection of top/latest songs from a certain artist/band. Since the last.fm graph used in our experiments was abstracted from such playlists, we see paths corresponding to songs by the same artist. It would be interesting to see the paths that emerge from graphs abstracted from actual user listening history, which may potentially be more diverse. We leave such exploration for future work.

We analyzed the "topic paths" that represent flow of ideas in the Cora graph of topics extracted from research paper citations. The top path of $\ell = 3$ had the topics: "Cooperative Human Computer Interaction (HCI), Distributed Operating Systems (OS), Input, Output and Storage for Hardware and Architecture". It turns out that in this dataset, there are several papers about application like group chat and distributed white boards that involve collaborative computing. Such applications are concerned about both reliability of distributed servers and real time end user experience. Hence, the topic path obtained from the graph highlighted

this less obvious connection. Longer heavy paths also exhibit interesting patterns, for instance, this heaviest topic path of length 7: "Object Oriented Programming, Compiler Design, Memory Management in OS, Distributed OS, Cooperative HCI, Multimedia and HCI, Networking Protocols, Routing in Networks" corresponds to a surprising, and non-trivial flow of ideas across a chain of topics.

Quality of paths using the heuristic approach. Figure 9 shows the path weight obtained using HeavyPathHeuristic as ℓ increases, for $k = 1$. The exact result as obtained by HeavyPath is plotted for comparison, along with an "upper bound" that represents the best possible score of a path for a given ℓ as detailed in Section 5.3. As a baseline, we also plot the path resulting from a purely greedy approach that builds the heaviest path of length ℓ by starting from the heaviest edge and repeatedly following heaviest edges out of the end nodes. For the Cora dataset in Figure 9(a), the memory-bounded heuristic is no worse than 50% of the theoretically best possible path weight, and it finds heavier paths of a given length compared with the greedy approach. Recall, the theoretically best possible path weight is estimated on the upper bound U_ℓ as explained in Section 5.3. The approach can be used to find a path connecting all nodes in the graph ($\ell = 69$). The results for the last.fm dataset in Figure 9(b) show that the heuristic approach is able to compute paths of long lengths, with the accuracy decreasing with path length. The greedy approach stalled after $\ell = 40$. The last.fm dataset has 6534 components, while other datasets have a single connected component. When the greedy starts with the heaviest edge, and the component to which it belongs has fewer than 40 nodes, the greedy algorithm restarts the process with the next heaviest edge. This results in a lot of exploration before the algorithm can result in a heavy path of $\ell > 40$. The heuristic manages to find paths of length up to 100. Even at that length, weight of the path it finds is about 37.5% of the theoretically best possible. Interestingly, the heuristic approach finds paths whose weight is very close to that of the best possible, for the Bay dataset as seen in Figure 9(c).

Figure 9(d) shows the empirical approximation ratio ρ as the path length increases, keeping the allocated memory fixed (as with Figure 9(a)) at 12GB. As the path length increases, ρ decreases. This is expected, given the same bootstrap from the exact algorithm, the estimate of the heuristic worsens with respect to the best possible weight. It should be noted that *ρ is not a ratio w.r.t. the optimal solution, instead a conservative estimate of the worst case ratio of the result of our algorithm on a given graph instance.*

Memory for the heuristic approach. When the path length is fixed, and the memory allocated to the heuristic approach is increased, ρ increases. Figure 9(e) illustrates this for the Cora dataset, with $\ell = 25, k = 1$ and memory allocation varied to store 10 to 1 million paths. It is worth noting that even with as little memory as needed to hold $5,000$ paths, ρ is already at 0.6 and with $250,000$ paths, it reaches 0.7. Further improvement is limited.

Running time for the heuristic approach. Figure 9(f) shows the running time of the heuristic as well as of HeavyPath and Greedy on the last.fm dataset. HeavyPathHeuristic takes negligible amount of time for post-processing, for paths of long length (shown up to $\ell = 100$). Greedy does not scale beyond $\ell = 40$. The other datasets show similar patterns for HeavyPath and HeavyPathHeuristic, while Greedy takes only tens of milliseconds on other data sets.

6.3 Discussion

We observed in Figures 7(a) that for one parameter setting of $\ell = 3$ on the Cora dataset, Rank Join outperformed HeavyPath in terms of running time for finding the top-1 path. We found that the random accesses performed for extending paths ending in high degree nodes resulted in a slower termination. For instance, consider the sub-graph of the graph in Figure 3 induced by the nodes $a', b', c', d'_1, \ldots, d'_n$, but with the following edge weights: $w_{(a',b')} = 1, w_{(b',c')} = 1, w_{(c',d'_1)} = 1$, and all other edge weights to nodes $d'_2 \ldots d'_n$ the same at 0.01. Rank Join would scan the 3 top edges and terminate (since the top path weight equals the threshold), while HeavyPath will scan $n - 1$ additional edges by performing random accesses. A potential enhancement to HeavyPath is to perform the random accesses in a "lazy" fashion. In particular, perform random access on demand, and in a sorted order of non-increasing edge weights. Such an approach would avoid wasteful random accesses to edges that have very low weight.

While our exact algorithms already scale much better than the classical Rank Join approach, our heuristics take a only tens of milliseconds to compute results that are no worse than 50% of the optimal for path lengths up to 50 on all datasets we tested. It is worth noting that heuristics for the problem of TSP have been extensively studied[10] for decades. Although these are not directly applicable to HPP, it would be interesting to explore the ideas employed in those heuristics to possibly get higher accuracy.

We conducted various additional experiments focusing on metrics such as number of edge reads performed, number of paths constructed, and the rates at which the termination thresholds used by the algorithms decay. The details can be found in Appendix B.

7. RELATED WORK

The problem of finding heavy paths is well suited for enabling applications as described in Section 1. Recently, there has been interest in the problem of itinerary planning [3, 18]. Choudhury et al. [3] model it as a variation of the orienteering problem, and in their setting the end points of the tour are given, making the problem considerably simpler than ours. Xie et al. [18] study it in the context of generating packages (sets of items) under user-defined constraints, e.g., an itinerary of points of interest with 2 museums and 2 parks in London. However, the recommendations that are returned to the user are sets of items which do not capture any order. In both these papers, the total cost of the POI tour is subject to a constraint and the objective is to maximize the "value" of the tour, where the value is determined by user ratings. In contrast, by modeling itinerary finding as a HPP problem, we aim to minimize the cost of the tour (through high value items), which is technically a different

[10] `http://www2.research.att.com/~dsj/chtsp`

problem. Another related work is [4] on generating a ranked list of papers, given query keywords.

In [8], Hansen and Golbeck make a case for recommending playlists and other collections of items. The AutoDJ system of Platt et al. [15] is one of the early works on playlist generation that uses acoustic features of songs. There is a large number of similar services – see [5] for comparison of some of these services. To our knowledge, playlist generation as an optimization problem has not been studied before.

The HPP was studied recently by [2] for the specific application of finding "persistent chatter" in the blogosphere. Unlike our setting, the graph associated with their application is ℓ-partite and acyclic. As mentioned in the introduction, adaptations of their algorithms to general graphs lead to rather expensive solutions.

As mentioned earlier, the HPP problem can be mapped to a length restricted version of TSP known as ℓ-TSP, where ℓ is the length of the tour. ℓ-TSP is NP-hard and inapproximable when triangular inequality doesn't hold [1]. For the special Euclidean case, there is a 2-approximation algorithm due to Garg et al. [6]. We propose a practical solution based on the well established Rank Join framework for efficiently finding the exact answer to this problem for reasonable path lengths. To the best our knowledge, this solution is novel.

Rank Join was first proposed by Ilyas et al. [10, 11, 13] to produce top-k join tuples in a relational database. They proposed the logical rank join operator that enforces no restriction on the number of input relations. In addition, they describe HRJN, a binary operator for two input relations, and a pipelining solution for more than two relations since a multi-way join is not supported well by traditional database engines. It is well known that the multi-way Rank Join operator is instance optimal, however similar optimality results are not known for iterated applications of the binary operator [17, 16]. Therefore our comparison of HeavyPath with the multi-way Rank Join approach, to establish both our theoretical and empirical claims, is well justified.

The seminal work on Rank Join [10] already mentions the potential usefulness of random accesses, however it does not evaluate it theoretically or empirically. A recent study [14] proposes a cost-based approach as a guideline for determining when random access should be performed in the case of binary Rank Join. As such, their results are not directly applicable for HPP. In contrast to all the prior work on Rank Join, we address the specific question of finding the top-k heaviest paths in a weighted graph and adapt the Rank Join framework to develop more efficient algorithms. Our techniques leverage the fact that the join is a self-join, store intermediate results and use random accesses to aggressively lower the thresholds and facilitate early termination. Besides, we develop techniques for carefully managing random accesses to minimize duplicate derivations of paths. Furthermore, we establish theoretical results that are in favor of using random access for repeated binary rank self-joins.

8. CONCLUSIONS AND FUTURE WORK

Finding the top-k heaviest paths in a graph is an important problem with many practical applications. This problem is NP-hard. In this paper we focus on developing practical exact and heuristic algorithms for this problem. We identify its connection with the well-known Rank Join paradigm and provide insights on how to improve Rank Join for this special setting. To the best of our knowledge, we are the first to identify this connection. We present the HeavyPath algorithm that significantly improves a straightforward adaptation of Rank Join by employing and controlling random accesses and via more aggressive threshold updating. We propose a practical heuristic algorithm which is able to provide an empirical approximation ratio and scales well both w.r.t. path length and number of paths. Our experimental results suggest that our algorithms are both scalable and reliable. Our future work includes improving memory usage and scalability of our exact algorithms, exploration and adaptation of ideas in [14] for improving the performance further, and application of our algorithms in recommender systems and social networks. It is also interesting to investigate top-k algorithms with probabilistic guarantees for the heavy path problem.

9. REFERENCES

[1] S. Arora and G. Karakostas. A $2 + \epsilon$ approximation algorithm for the k-MST problem. *Math. Prog.*, 2006.

[2] N. Bansal, F. Chiang, N. Koudas, and F. W. Tompa. Seeking stable clusters in the blogosphere. In *VLDB*, 2007.

[3] M. Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu. Automatic construction of travel itineraries using social breadcrumbs. In *HT*, 2010.

[4] M. D. Ekstrand, P. Kannan, J. A. Stemper, J. T. Butler, J. A. Konstan, and J. Riedl. Automatically building research reading lists. In *RecSys*, 2010.

[5] B. Fields, C. Rhodes, and M. d'Inverno. Using song social tags and topic models to describe and compare playlists. In *Workshop on Music Recommendation and Discovery*, 2010.

[6] N. Garg. Saving an epsilon: a 2-approximation for the k-mst problem in graphs. In *STOC*, 2005.

[7] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB*, 2005.

[8] D. L. Hansen and J. Golbeck. Mixing it up: recommending collections of items. In *CHI*, 2009.

[9] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. 10(1), 1962.

[10] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.

[11] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, 2004.

[12] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, 2009.

[13] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.

[14] D. Martinenghi and M. Tagliasacchi. Cost-aware rank join with random and sorted access. *TKDE*, 2011.

[15] J. C. Platt, C. J. C. Burges, S. Swenson, C. Weare, and A. Zheng. Learning a gaussian process prior for

automatically generating music playlists. In *NIPS*, 2001.
[16] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, 2008.
[17] K. Schnaitter and N. Polyzotis. Optimal algorithms for evaluating rank joins in database systems. *TODS*, 2010.
[18] M. H. Xie, L. V. S. Lakshmanan, and P. T. Wood. Breaking out of the box of recommendations: from items to packages. In *RecSys*, 2010.

APPENDIX

A. DYNAMIC PROGRAMMING DETAILS

Essentially, dynamic programming constructs all simple paths of length ℓ in order to find the heaviest among them. But unlike DFS, it aggregates path segments early, thus achieving some pruning. More precisely, here are the equations of the dynamic program. Letters i, j, y denote variables which will be instantiated to actual graph nodes when the program is run. S denotes the avoidance set. The variable l will be instantiated in the range $[2, \ell]$.

$$P^l_{i,j,S} = \text{MAX}\{P^{l-1}_{i,y,S\cup\{y\}} \circ e(y,j) \mid (y,j) \in E, \\ y \notin S, j \in S\}$$

$$P^1_{i,j,S} = \begin{cases} \text{CREATE}(P, e(i,j)) & \text{if}(i,j) \in E \\ \text{NULL} & \text{otherwise} \end{cases}$$

In the above equations, we can think of $P^l_{i,j,S}$ as a "path object" with properties *path* and *weight*. Here, $P^l_{i,j,S}.path$ denotes the heaviest S-avoiding path from i to j of length l, and $P^l_{i,j,S}.weight$ denotes its weight. The operator MAX takes a collection of path objects and finds the object with maximum weight among them. The "$\circ$" operator takes a P object and an edge $e(u,v)$, concatenates the edge with the *P.path* and updates *P.weight* by adding the weight of the edge. Finally, $create(P, e(i,j))$ creates a path object P whose *path* property is initialized to (i,j) and whose *weight* property is initialized to the edge weight of (i,j).

We invoke the dynamic program above using $P^\ell_{\$i,j,\{j\}}$ every node $j \in V$, where we have left the start node as a variable $\$i$. The heaviest path of length ℓ in the graph is the heaviest among the paths found above for various j.

For finding top-k heaviest paths of a given length for $k > 1$, all we need to do is define the MAX operator so that it works in an iterative mode and finds the next heaviest path segment (ending at a certain node, of a given length, and avoiding certain set of nodes). We can apply this idea recursively and easily extend the dynamic program for finding top-k heaviest paths of a given length.

B. ADDITIONAL EXPERIMENTS

Our main result is that HeavyPath is algorithmically superior to all compared algorithms including Rank Join, in terms of the amount of work done. We establish this claim in multiple levels, by performing additional experiments that include comparing the number of edges read, the paths created, and the rate at which the thresholds decay. In all these experiments, we set $k = 1$, i.e., we focused on the top path.

Edge Reads. We measured the total number of edge reads performed by HeavyPath and by NextHeavyPath, where every time an edge is read, under sorted or random access or while performing joins to construct paths, is counted as one. We show the results in Figure 10. The # edge reads for NextHeavyPath exceeds 10^{10} for path lengths beyond 4 on Cora and last.fm and beyond 11 on Bay, which then runs out of memory. On the other hand, HeavyPath is able to go up to lengths 8, 7, and 36 on the three respective data sets, incurring far fewer edge reads. The reason for running out of memory has to do with the number of paths computed and examined, investigated next.

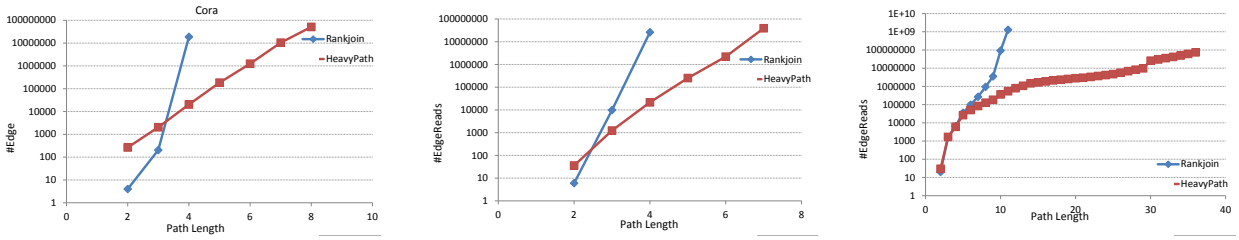
Paths Constructed. We counted the number of paths. In case of Rank Join, we only counted paths of length ℓ where ℓ is the required path length. In case of HeavyPath, we counted paths of all lengths $\leq \ell$ that are constructed and stored by it in its buffers. The results are shown in Figure 13. The relative performance of HeavyPath and Rank Join w.r.t. the number of paths computed by them is consistent with the trend observed in the previous experiment on # edge reads, showing HeavyPath ends up doing significantly less work than Rank Join and hence is able to scale to longer lengths.

Threshold Decay. Both algorithms rely on their stopping threshold, θ for Rank Join and θ_ℓ for HeavyPath for early termination. We measured the rate at which the thresholds decay since that gives an indication how quickly the algorithm will terminate. Figure 11 shows the results. For each algorithm, the threshold value is shown until the algorithm terminates, successfully finding the heaviest path. It is obvious that in all cases, the threshold θ_ℓ of HeavyPath decays extremely fast, whereas in comparison, the threshold θ used by Rank Join decays much more slowly. Since HeavyPath employs random accesses whereas Rank Join doesn't, it is clear that random access is responsible for the rapid decay of the threshold, resulting in significant gain in performance. Please notice, the three experiments above are absolutely unaffected by systems issues, including garbage collection.

Running time. Finally, we separated the total running time into time spent on garbage collection and the rest.[11] Let's call the rest "compute time" for convenience. We compare the *total time* for HeavyPath with the *compute time* for the other algorithms. It is worth noting here that for the majority of the points reported in our results (which of course correspond to those cases where the appropriate algorithm terminated) successfully, garbage collection time was either 0 or very small. The result is shown in Figure 12. For simplicity, we "time out", i.e., stop the experiment, whenever the time taken exceeds 10^6 seconds. It can be observed that despite taking out the (small to negligible amount of) garbage collection from the total times of other algorithms, the total time of HeavyPath is still significantly less than the compute times of other algorithms.

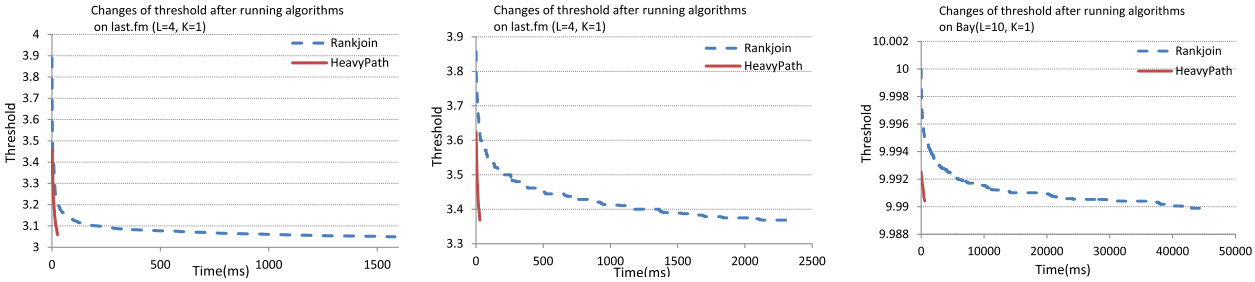
In each of these experiments, we can observe that HeavyPath significantly outperforms other algorithms. These ex-

[11] We did this using the *Jstats* tool and would be happy to provide additional details on this if required.



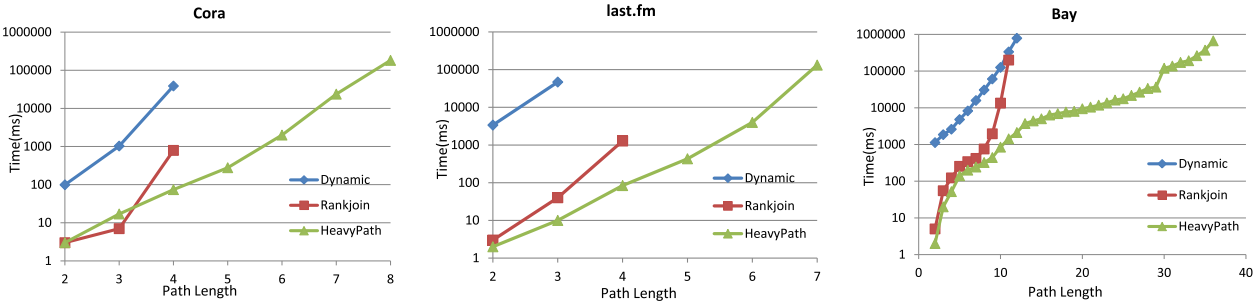
(a) Cora. (b) last.fm. (c) Bay.

Figure 10: Comparing the number of Edge Reads.



(a) Cora. (b) last.fm. (c) Bay.

Figure 11: Comparing Threshold Decays of Rank Join and HeavyPath



(a) Cora. (b) last.fm. (c) Bay.

Figure 12: Comparing total time of HeavyPath to compute time of other algorithms

periments clearly establish that HeavyPath's superiority over the other algorithms in terms of the amount of work done, i.e., in purely algorithmic terms, regardless of systems issues, and that our findings are not in any way compromised by garbage collection.

HeavyPath

Cora(k=	L=1	L=2	L=3	L=4	L=5	L=6	L=7	L=8	Total
L=2	3	263	x	x	x	x	x	x	266
L=3	12	1101	931	x	x	x	x	x	2044
L=4	36	3338	14916	2032	x	x	x	x	20322
L=5	74	6816	132159	40178	5075	x	x	x	184302
L=6	231	19944	707158	414338	94242	8875	x	x	1244788
L=7	1580	74406	3382794	4964856	1720775	271104	17558	x	10433073
L=8	1580	74406	3382794	24279038	18017155	4743073	564389	22626	51085061

last.fm (	L=1	L=2	L=3	L=4	L=5	L=6	L=7	Total
L=2	6	30	x	x	x	x	x	36
L=3	32	419	791	x	x	x	x	1242
L=4	202	2970	14884	3536	x	x	x	21592
L=5	1095	17099	167155	51787	15475	x	x	252611
L=6	5648	102437	1049963	764093	231408	55840	x	2209389
L=7	44422	1241344	25537137	5809867	4840185	1485657	288439	39247051

Bay (k=1	L=1	L=2	L=3	L=4	L=5	L=6	L=7	L=8	Total
L=2	7	23	x	x	x	x	x	x	30
L=3	400	1137	142	x	x	x	x	x	1679
L=4	1446	4065	471	47	x	x	x	x	6029
L=5	6038	17140	3073	243	25	x	x	x	26519
L=6	10828	30711	8174	748	123	30	x	x	50614
L=7	16911	48658	16116	1700	255	73	37	x	83750
L=8	24476	70845	27695	3391	551	136	61	28	127183

(a) HeavyPath

Rankjoin

Cora(k=1)	#Paths
L=2	2
L=3	68
L=4	4676135

last.fm(k=1)	#Paths
L=2	3
L=3	3280
L=4	6533631

Bay (k=1)	#Paths
L=2	1
L=3	19
L=4	105
L=5	1713
L=6	10118
L=7	44004
L=8	226693

(b) Rank Join

Figure 13: Comparing the number of paths created by HeavyPath and Rank Join